1er cours

Rappels de mias-1 (algo en Pascal, sans pointeurs et sans recursion et sans complexite),
et annonce pour mias2 du propos :pointeurs recursion complexite types abstraits et implementation
listes files piles, arbres binaires et arbres generaux.
Suivi d'un exemple en pseudo-langage proche du pascal (mais en fancais)

(fin 1er cours)

2eme cours

une simulation de l'exemple en
insistant
sur l'allocation de memoire lors de la compilation pour les variables globales
sur l'allocation (sur le tas) lors de l'appel pour les variables locales et arguments passes par valeur dans les fonctions et procedures (et la desallocation a la sortie
de celles-ci) histoire de preparer a la recursion

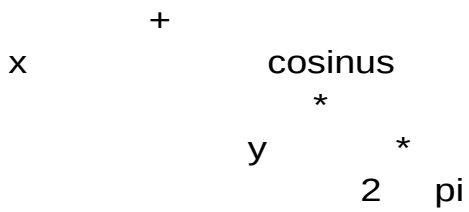
complements sur
les structures de controles:
---
tantque C faire A ftq
repeter A jusqua C
pour i <-- ideb a ifin repeter A finpour
si C alors A1 {sinon A2} finsi
selon v =
v1 : A1
v2 : A2

{autrement Adefaut}
finselon
---

les types elementaires : reel, entier, booleen, text, caractere
les types composes: enregistrements, tableaux uni et multidimensionnels
(j'ai precise qu'en general le dernier indice correspond aux emplacements
contigus en memoire : T[i,j+1] jouxte T[i,j])
type file of element

les actions elementaires :
affectation et appel de procedure

la notion d'expression et d'evaluation d'expression :
evaluation des arguments dabord puis application de l'operateur ou de
la fonction. (cas particulier de ET et OU : dans e1 ET e2, si e1 est faux,
e2 n'est PAS evalue}
representation d'un arbre d'expression evalue en profondeur a gauche dabord:
x+(cosinus(y*(2*pi)) :



puis fonctionnement des appels des fonctions et procedures:
0) les variables locales masquent les variables globales alias
1) a l'appel de Truc, les variables locales (et arguments passes par
valeurs) recoivent
de la memoire qui disparait au retour
2) lorsque PREtruc appelle truc, les variables locales (et arguments
passes par valeurs)
de PRETruc et le point de retour sont ranges dans un "environnement"
qui est empile dans une pile allouee par le systeme, lors du retour cet
environnement est depile.
Illustration sur une fonction TrioisFoisqui appelle DeuxFois qui elle meme
appelle
 UneFois (en vue d'approcher la recursion en douceur):
TroisFois(n) = n*DeuxFois(n-1)

DeuxFois(n) = n* UneFois(n-1)

UneFois(n) = n

donc TroisFois(n) = n*((n-1)*(n-2))

```
fonction UneFois(n:entier):entier
var resultat :entier
debut
resultat<-- n
retourner(resultat)
fin

fonction DeuxFois(n:entier):entier
var resultat :entier
debut
```

```
resultat <-- n*UneFois(n-1)
retourner(resultat)
fin

fonction TroisFois(n:entier):entier
var resultat :entier
debut
resultat <-- n*DeuxFois(n-1)
retourner(resultat)
fin
```

detail donc : on empile
l'environnement de (inconnu))
lors de l'appel de TroisFois(5)
l'environnement (n=5 , resultat =?) de TroisFois(5)
lors de l'appel de DeuxFois(4)
l'environnement (n=4 , resultat =?) de DeuxFois(4)
lors de l'appel de UneFois(3)

puis on depile
l'environnement (n=4 , resultat =?) de DeuxFois(4)
lors du retour depuis UneFois(3)(renvoie 3)
l'environnement (n=5 , resultat =?) de TroisFois(5)
lors du retour depuis DeuxFois(4)(renvoie 12)
l'environnement de (inconnu))
lors du retour depuis TroisFois(5)(renvoie 60)

la version recursive qui generalise ceci est

```
fonction LFois(n:entier; profondeur:entier):entier
var resultat :entier
debut
si profondeur =1 alors
        retourner n
sinon
        resultat <-- n*LFois(n-1, profondeur-1)
        retourner(resultat)
finsi
fin
```

elle est egalement simulee ainsi que les operations d'empilements et
depilements des environnements

---

plusieurs exemples de recursion et raisonnemments recursifs:
factorielle avec une preuve : fact(n) = n! au rang 0 et si c'est vrai au rang i c'est vrai au rang i +1
donc c'est vrai pour n>=0,
MAIS il faut aussi s'assurer que la recursion termine, exemple

a fois b = a fois (b+1) - a
a fois 0 = 0

le pb ici est que fois est correct mais ne termine que pour les nombres negatifs,

autre exemple (dont les empilements sont simules egalement) : la procedure envers
(lit une chaine (terminee par un #) et l'ecrit a l'envers sans la rangee dans un tableau
(les valeurs successives des caractereslus sont empiles lors des appels puis affiches au retour)

enfin fib(n) = fib(n-1)+fib(n-2) et fib(0)=fib(1) =1

cette double recursion m'a permis de montrer un arbre d'appels,

puis une autre version de fib constitue de fibMain (non recursive mais qui met a 0
un tableau de N valeurs avant d'appeller fibrec(N) recursive semblable a fib
mais lorsqu'elle est appellee (fibrec(i))commence par verifier que en tab[i] on a une valeur nulle,
si tab[i] <>0 alors on renvoie tab[i] sinon on effectue les appels recursifs PUIS on range le resultat de fibrec[i] avant de le renvoyer, dans T[i]...

ainsi on passe d'une complexite exponentielle (pas dit en ces termes) a une complexite lineaire puisque
l'arbre est un peigne et ne calcule qu'une fois chaque valeur....

(ca me semble important qui'il constante que les doubles appels peuvent etre catastrophiques,
mais que la memorisation peut sauver la mise...

fibonacci en detail plus les tours de hanoi (sans simulation)
plus le pb de la distance d'edition (cout minimum pour le passage d'une chaine a une autre

par insertions deletions substitution par programmation dynamique, qui s'exprime facilement
de maniere recursive et devient en n2 quand on memorise les etapes internmediaires)

plus precisement :
- la fin de la memorisation dans un tableau des resultats intermediaires de fibonacci recursive
qui fait passer (evoque sans details) d'une fonction recursive exponetielle a une fomction recursive lineaire,
en transformant l'arbre des appels en un peigne.
- les tours de hanoi (affichage "X-->Y" des deplacements des n pierres sur la tige de depart D vers la tige
vide d'arrivee A en utilisant une tige intermediaire I.
- calcul de la distance d'edition entre deux chaines (nombre minimal d'operations d'insertions deletions
substitutions permettant de passer de la premiere chaine a la seconde). Ce nombre minimal correspond a un
meilleur chemin dans une grille entre le debut (vide,vide) et la fin (longueur chaine1, longueur chaine2).
Le raisonnement est recursif:la position d'arrivee (longueur chaine1, longueur chaine2) peut etre atteinte de
trois manieres a partir de l'etape precedente selon que l'on vienne de gauche (insertion), du haut(deletion)
ou de la diagonale (substitution). On fait dans chacun des trois cas la somme du chemin partiel entre le debut
et cette position et du cout du dernier deplacement dans ce cas, et on prend le min. (je peux vous passer un
papier detaille). La aussi la memorisation dans un tableau permet de se ramener a un cas polynomial (ici en
longueur chaine1 x longueur chaine2) alors que la version naive est exponentielle.

Je n'ai pas fini ce dernier point (on donne cette solution en general comme etant de la programmation
dynamique)

-----(document détaillé disponible la-dessus)-------

-complexite : introduction, notion de domination asymptotique (O) et de
meme ordre de grandeur asymptotique (TETA), rappel de "equivalence a
l'infini" et proprietes diverses, Pire des cas, cas moyen et meilleur des
cas

----(document détaillé disponible la dessus)-------

---
complexite suite:
- quelques regles pour le pire des cas: enchainement, conditionnelle, et
ordre d'evaluation des procedures/fonctions, exemples a l'appui...
puis le cas recursif, avec:

1) factorielle
2) fonct(n,c1,c2) = fonc(n/2,c1,c2)+fonc(n/2,c2,c1) avec fonc(1,c1,c2)=c1/c2
dans le cas n=2puissance(p) (* teta(n) *)
3) le cas de la dichotomie, avec separation par milieu par defaut(a+b div2)
et decoupage en (a,milieu) et (milieu+1,b), avec pour commencer le cas n=2puissance(p) (ou on arrvive facilement au theta(log(n)))
puis le cas general qui est obtenu en supposant
1) T est croissante
2) 2puissance(p-1) < n <= 2puissance(p) = n' (p est alors le plafond de log(n))

on encadre alors T(n) par T(2puissance(p)) et T(2puissance(p-1))
et on montre ainsi par majorations et minorations que T(n) est en theta(log(n))

Ensuite les pointeurs, les listes, files, piles

----(document détaillé la dessus)-------

Ensuite les arbres généraux avec leur implémentation en filsdegauche-frerede droite

ci-dessous un exo:

ecrire une procedure qui duplique le sous-arbre de racine a1 en un sous-arbre de racine a2 (a2 n'a pas de valeur particuliere au depart).
(passage par adresse dans une procedure recursive, et plus
precisement raccrochage des enregistrements)

solution proposee:

copieArbre(a1:Pnoeud ; var a2 : Pnoeud)

var ...

```
debut
si a1=nil alors
        a2 <--nil
sinon
        new(a2)
        a2^.val <-- a1^.val
        a2^.freredeD<--nil
        CopieArbre(a1^.filsdeG, a2^.filsdeG)
        y1<--a1^.filsdeG
        y2<--a2^.filsdeG
        Tantque y1 <>nil faire
                copierarbre(y1^.freredeD, a2^.freredeD)
                y1<--y1^.freredeD
                y2<--y2^.freredeD
        finTq
finsi
fin
```

la difficulte est les appels comme "CopieArbre(a1^.filsdeG, a2^.filsdeG)"
qui permettent la connection directe du sous arbre nouvellement cree a son antecedent (ici son pere)